The Model Context Protocol (MCP): Enabling Secure, Scalable Agent-Tool Interactions

Executive Summary

The Model Context Protocol (MCP) is an emerging open standard designed to streamline how AI agents (powered by large language models, LLMs) interact with external tools and data sources. By defining a consistent protocol for tool-use, MCP aims to be a "USB-C port" for AI applications – a universal interface that allows AI agents to plug into various services securely and seamlessly. This white paper provides a formal analysis of MCP targeted at engineers in security and AI, covering its core concepts, the problems it addresses in agentic AI workflows, the current security model and its gaps, and the future directions for making MCP more robust. Key takeaways include:

- **Deterministic Tool Invocation:** MCP replaces brittle prompt-based APIs with deterministic, well-defined function calls, reducing the trial-and-error currently required for agents to use tools (Securing the Model Context Protocol | codename goose). This improves reliability and predictability when LLM-based agents execute actions on external systems.

- **Integration at Scale:** MCP offers a vendor-neutral, standardized way to integrate countless tools and data sources with AI. Early adoption by major platforms (e.g. Microsoft's Copilot, Cloudflare Workers) underscores MCP's potential to simplify generative AI development by treating tool integrations as interchangeable modules (MCP: The missing link for agentic AI?) (MCP: The missing link for agentic AI?). As one Microsoft team noted, connecting to an MCP server automatically augments an agent's capabilities and reduces maintenance overhead (MCP: The missing link for agentic AI?).

- **Security Considerations:** While MCP's design leverages established protocols (JSON-RPC over transports like HTTP and STDIO), its initial specification left security to the transport or implementer. This has exposed gaps such as **lack of**

built-in authentication or fine-grained authorization, potential data leakage to third-party LLMs, and supply chain risks from unvetted third-party MCP servers (MCP my HTTPS! · A guy with 'Ego' in his name) (Securing the Model Context Protocol | codename goose). A current analysis finds that **access control is only at the session level** and sensitive actions are not sufficiently gated, which is insufficient for enterprise use.

- **Future Directions:** Ongoing efforts are addressing these gaps. The community is adding **identity and authentication mechanisms** (OAuth flows to identify users, agents, and devices (Securing the Model Context Protocol | codename goose) (Securing the Model Context Protocol | codename goose)), exploring **streaming-friendly protocols** (e.g. WebSockets or gRPC for bi-directional communication beyond HTTP/SSE), improving **tool permission metadata** (marking actions as read-only or destructive to enable human approval workflows (Securing the Model Context Protocol | codename goose)), and mitigating **supply chain risks** (through code signing, trusted registries, and allow-lists for MCP extensions (Securing the Model Context Protocol | codename goose)). These enhancements aim to evolve MCP into a secure, enterprise-grade interface for autonomous AI-agent integrations.

In the sections that follow, we delve into MCP's architecture and role, how it tackles core challenges in agentic AI, the current security landscape with identified weaknesses, and the roadmap for strengthening the protocol. Technical examples and expert insights are included to illustrate the concepts.

Introduction to the Model Context Protocol (MCP)

Model Context Protocol (MCP) is an open protocol introduced by Anthropic in late 2024 to standardize how AI models connect to external data sources and tools (MCP: The missing link for agentic AI?). In essence, MCP defines a common format and procedure for an AI **agent (client)** to invoke operations on an external **tool or service (server)** in a controlled way. Anthropic describes MCP as *"an open standard that enables developers to build secure, two-way connections between their data sources and AI-powered tools"* (MCP: The missing link for agentic AI?). By providing a uniform interface, MCP is intended to do for AI-agent integrations what traditional APIs did for web services – enable any AI application to talk to any service using a predictable, well-defined contract (MCP: The missing link for agentic AI?).

At its core, MCP formalizes the interaction between an agent and a tool as a remote procedure call using JSON messages. In fact, *"all transports use JSON-RPC 2.0 to*

exchange messages" according to the specification (MCP my HTTPS! · A guy with 'Ego' in his name). The MCP protocol defines a set of standard **message types** (for requesting a tool invocation, returning a result, reporting errors, etc.) and allows these messages to be carried over different **transport layers**. Currently, two primary transports are defined: a local **STDIO transport** (where the agent and tool run on the same host and communicate via standard input/output streams) and an **HTTP Server-Sent Events (SSE) transport** for networked scenarios (MCP my HTTPS! · A guy with 'Ego' in his name). Both approaches support streaming responses (important for long-running tool actions that yield incremental output) but differ in deployment: STDIO is simple and low-latency for local tools, whereas HTTP/SSE allows the MCP server to run remotely (e.g. as a web service).

MCP Servers in this context are not servers in the traditional sense of hosting a standalone service. Rather, an "MCP server" is essentially an adapter or **shim layer** that wraps an existing tool, API, or system, exposing a set of well-defined actions to the agent (Securing the Model Context Protocol | codename goose). The MCP server translates the agent's high-level requests into the actual API calls or commands needed to perform the task (e.g. calling a REST endpoint, executing a CLI command, or invoking a function in a SDK). As Block's engineers describe, these servers *"act as a client layer (either locally or remotely) to help the agent proxy function calls to an existing service, tool, API or RPC in a deterministic manner"* (Securing the Model Context Protocol | codename goose). In other words, the MCP server plays the role of a **client library** for the underlying service, but one that the AI agent can control via a standard protocol.

To illustrate how MCP works, consider a user asking an AI agent to perform a multi-step task such as *"clone the repository `block/goose` from GitHub."* The agent would load an appropriate **Git integration tool** via MCP, which provides a function (e.g. `clone_repository`) to perform git clone operations. The agent calls this function through MCP, supplying parameters like the repository URL or name. The MCP server (Git tool adapter) receives the JSON-RPC request and executes the actual `git clone` command (perhaps via a GitHub API or shell command), then returns a result back to the agent. This interaction is diagrammed in **Figure 1**, showing how the human's request flows through the agent and MCP server to the external Git system, and the results flow back:

(Securing the Model Context Protocol | codename goose) *Figure 1: High-level MCP workflow. The agent loads available MCP tools (e.g. a Git integration providing a `clone_repository` action). When the human requests a repo clone, the agent invokes `clone_repository` via MCP. The MCP server translates this into the appropriate Git command (`git clone`), performing the action on the external service and returning the outcome to the agent.* (Securing the Model Context Protocol | codename goose)
(Securing the Model Context Protocol | codename goose)

MCP thus establishes a clear separation of concerns:

- The **Agent** collects the tools exposed by the MCP server and sends those descriptions with the messages to the LLM. If the LLM determines a tool execution is needed, it'll specify the parameters in its response to the Agent, who then delegates the tool execution to the MCP Server.

- The **MCP Server** both packages the tools and encapsulates *how* to perform those actions. Through the specification, it ensures the agent doesn't need to discover or learn the tool's interface through trial and error.

- The role of the **Underlying Service** (e.g. GitHub, Jira, a database, etc.) remains unchanged; MCP simply provides a safe conduit for the agent to use it.

This approach has a vital benefit: the agent's interface to all tools becomes consistent. No matter what the tool is – a SaaS API, a local CLI, or a cloud database – the agent interacts with it via the same MCP protocol (JSON-RPC calls over an available transport). This uniformity is why MCP is often described as *"AI's version of the API"*, standardizing how software (in this case an AI) talks to other software (MCP: The missing link for agentic AI?). In the next section, we examine the specific challenges in agentic AI that MCP is intended to solve and how it addresses them.

Problems MCP Aims to Solve in Agentic AI

Agentic AI refers to AI systems (typically powered by LLMs) that can autonomously take actions to accomplish goals, often by interacting with external tools or applications. Prior to MCP, integrating an AI agent with various tools was ad-hoc: developers either hard-coded API calls the model could trigger, or more commonly, tried to have the LLM *infer* how to use a tool from its documentation or examples. This often leads to unpredictable behavior, as the agent might guess incorrect commands or require prompt engineering tricks to use a tool reliably. MCP directly tackles several of these pain points:

- **Deterministic Tool Invocation:** One of MCP's primary goals is to eliminate the uncertainty of letting an AI figure out tool usage from natural language. Instead, the agent is given a deterministic, code-level interface for each action. or example, if the agent needs to retrieve a user's data from a database, an MCP server might provide a tool called "get_user" and indicate it requires a parameter of "user_id". The agent can simply invoke the "get_user" tool, passing along a "user_id" of "alice" to get a user record back. As Block's security team notes, this yields *"well defined interfaces that reduce the amount of experimentation/brute force required for agents to perform helpful tasks."* (Securing the Model Context Protocol | codename goose) The agent doesn't have to struggle with *"how to interact with*

Jira, GitHub and the Git CLI" in the middle of solving a problem (Securing the Model Context Protocol | codename goose); those interactions are handled correctly by the MCP server. This determinism makes agent behavior more predictable and safer, as the range of possible actions is constrained to the MCP tool API.

- **Integration Scalability and Reusability:** MCP provides a **standard integration layer** that tool developers can implement once and make available to any AI agent. This greatly improves scalability of integrations. Instead of every AI application writing its own custom code (or prompts) to use, say, Gmail or Salesforce, a single MCP server for Gmail can serve any agent that speaks MCP. Already, the ecosystem is growing rapidly – by early 2025 there were *"3,000 and counting"* MCP servers published by the community for all manner of services and utilities (Blog | codename goose). Major tech companies have started to embrace MCP to avoid reinventing the wheel. For instance, Microsoft announced support for MCP in its Copilot platform, so third-party MCP tools can be used in Copilot-enabled workflows (MCP: The missing link for agentic AI?). Cloudflare's recent introduction of managed MCP servers on their cloud platform further indicates that *"vendors are jumping on board"* to make AI-agent integration easier and more portable (MCP: The missing link for agentic AI?) (MCP: The missing link for agentic AI?). The outcome is an **ecosystem of interchangeable tools**: an AI agent can gain new skills simply by connecting to a new MCP server (e.g. plug in a CRM integration to enable sales-support actions) – much like a human installing a browser extension to get new functionality.

- **Consistent Agent Interfaces:** By standardizing the format of tool interaction, MCP gives AI agents a uniform interface for all tools. Agents do not need custom logic for each integration; they all appear as callable functions or actions exposed via MCP. This consistency also simplifies the agent's internal design. For example, an agent can maintain a single planning and execution loop where it considers which MCP-provided action to invoke next, rather than juggling different integration mechanisms. From the developer's perspective, there is also a consistent pattern to add new capabilities: implement an MCP server (or use an existing one) and register it with the agent platform. Microsoft highlighted this benefit, stating that *"when connecting to an MCP server, actions and knowledge are automatically added to the agent and updated as functionality evolves. This simplifies the process of building agents and reduces time spent maintaining the agents."* (MCP: The missing link for agentic AI?) In short, MCP becomes a **common language** between AI agents and tools, allowing any tool to speak to any agent in a plug-and-play fashion.

To make this concrete, consider a snippet of an MCP integration. Below is an example (in Python) adapted from Block's engineering blog, showing how a developer can define a tool method that an agent could call via MCP:

```python
@mcp.tool()
async def submit_feedback(feedback: str) -> Dict[str, Any]:
    """Submit feedback to the Snowflake team.

    Args:
        feedback: Feedback message

    Returns:
        Dictionary containing feedback status
    """
    return snowflake_client.submit_feedback(feedback_text=feedback)
```

Listing 1: Example MCP tool function (Python). The `@mcp.tool()` decorator registers this function as an action that the agent can invoke via MCP. When called, it simply delegates to a native client (`snowflake_client`) to perform the real operation, here sending feedback to Snowflake. (Securing the Model Context Protocol | codename goose) (Securing the Model Context Protocol | codename goose)

By writing a small wrapper like the above, the developer makes a new capability ("submit feedback to Snowflake") available to any compliant AI agent. The agent doesn't need to know implementation details – it will see a tool named `submit_feedback` that requires a single parameter named `feedback`, relying on MCP to handle execution and return the result.

In summary, MCP addresses critical needs in agentic AI development by providing deterministic actions, an integration abstraction that scales, and consistent interfaces for complex tool ecosystems. These strengths, however, must be balanced with careful attention to security, which we analyze next.

Security Model and Current Landscape

Security is a paramount concern for MCP, given that it enables automated systems to perform actions that could potentially have real-world side effects. The current MCP

specification and early implementations have adopted some security practices but also reveal important gaps that need to be addressed for MCP to be safely deployed, especially in enterprise or sensitive environments.

Implicit Trust and the Transport Layer: Notably, the MCP core specification itself does **not mandate authentication or encryption at the protocol message level** – it largely delegates security to the chosen transport. For example, if using the HTTP/SSE transport, one would normally run it over HTTPS and could leverage HTTP headers or tokens for authentication, but MCP does not define a standard auth handshake within JSON-RPC. In fact, a skeptic observer critically summarized MCP as *"a protocol-not-protocol"* that *"allows LLMs to completely ignore decades of well thought-out APIs and instead force humans to write API wrappers and expose them via either unauthenticated STDIO or HTTP SSE without a single mention of the authentication methods"* (MCP my HTTPS! · A guy with 'Ego' in his name). This harsh assessment underscores that MCP's initial design prioritized connectivity and ease of use, assuming a level of trust in the environment or leaving authentication as an exercise for the developer. In practice, many MCP deployments to date have been **local or single-user scenarios** (running an agent and tool on a user's machine), where the security perimeter is the machine itself. In those cases, lack of protocol-level auth is acceptable. However, as MCP moves to multi-user, networked scenarios (e.g. a cloud-hosted MCP service), the absence of built-in authZ/authN becomes a glaring gap.

Lack of Fine-Grained Permissions: Closely related is the issue of authorization granularity. The current MCP session model is essentially all-or-nothing – if an agent is permitted to connect to an MCP server, it can invoke any tool that server offers. *"MCP lacks a built-in permissions model, so access control is at the session level — meaning a tool is either accessible or completely restricted,"* as noted by Yoko Li in a recent a16z analysis (MCP: The missing link for agentic AI?). For enterprise use, this is insufficient: one would want certain actions or data access to be limited based on user roles, policies, or contexts. Today, those checks must be implemented inside the MCP server logic or via external means. The absence of a standardized permission schema in MCP means inconsistent security – one MCP server might implement its own token scopes or API keys, while another simply trusts any connected agent. Until this is standardized, **tool integrations remain a weak link**, potentially exposing sensitive operations with inadequate checks. For example, a company might have an MCP server for internal HR data – ideally it should enforce that only an "HR agent" can call `get_salary_info`, whereas a "DevOps agent" should not even see that function. Currently, MCP does not provide that level of discrimination out-of-the-box.

Data Privacy and LLM Sandboxing: Another security concern arises from the fact that many AI agents rely on third-party LLM APIs (like OpenAI or Anthropic models) to drive their decisions. If an MCP tool returns sensitive data (customer PII, credentials, etc.), that

data might inadvertently be sent back to the LLM provider as part of the agent's next prompt or context, unless precautions are taken (Securing the Model Context Protocol | codename goose) (Securing the Model Context Protocol | codename goose). Block's security engineers highlight this scenario, noting that if, for instance, an MCP tool returns a Social Security Number from a secure database, *"you run the risk of that data being exposed to the underlying LLM provider"* if the agent naively includes it in subsequent prompts (Securing the Model Context Protocol | codename goose). Mitigations for this include **restricting which LLMs an agent can use with a given tool** – e.g. enforcing that certain tools can only be used by agents running on a self-hosted (local) model, or on specific approved LLM services (Securing the Model Context Protocol | codename goose). In concept, one could configure an MCP server with an allow-list of LLM backends permitted to call it. Another mitigation is to instruct the agent (or design the agent architecture) not to share certain tool outputs with other tools or external calls. These are emergent strategies, not standardized yet, indicating a current gap in how data confidentiality is maintained through the chain of agent reasoning.

Deterministic yet Unpredictable: It's worth noting a subtle security dynamic: MCP makes tool execution deterministic *given a particular agent action*, but the agent's decision to invoke a tool is driven by an LLM which can be nondeterministic or even adversarially manipulated. Developers are trained to never trust direct user input. While much of that validation often is skipped when coming from internal domains and APIs, data from LLMs needs to be treated much the same way as data coming from users. Put simply, don't blindly trust LLM output just as you wouldn't blindly trust user input. In effect, the **agent is a new kind of user** – one that can be tricked via prompt injection to misuse tools. The MCP protocol itself *"isn't implicitly allowing for agents to perform actions that users couldn't"*, but an agent might issue sequences of actions *"that users wouldn't choose"* (Securing the Model Context Protocol | codename goose). This means **the threat model must include the agent itself**. If an attacker can influence the agent's prompts (for instance via a poisoned input or a malicious response from another tool), they might induce the agent to call tools in harmful ways. An example is an agent with a file system MCP tool: normally it uses `read_file` and `write_file` for benign tasks, but a prompt injection could trick it into calling `delete_file('/important/data')`. The MCP server will dutifully execute that command unless safeguards are in place. Currently, safeguarding against such misuse is left to "best practices" – e.g. code reviews of what an MCP integration allows, setting conservative limits, or requiring user confirmation for dangerous actions. As we will see, upcoming features and patterns (like marking certain actions as destructive and requiring a human-in-the-loop) are aimed at closing this safety gap.

Transport Security and Unmentioned Alternatives: The choice of transport also plays a role in security. STDIO transport, while convenient for local use, is vulnerable if an attacker gains local access (they could potentially hijack or sniff the pipes). The

HTTP/SSE transport assumes the use of HTTPS for encryption and perhaps bearer tokens for auth, but again MCP doesn't specify the scheme. Interestingly, a WebSocket-based transport has been used in some implementations (e.g. experimental WebSocket support for streaming) but *"isn't even mentioned in the spec"* as of early 2025 (MCP my HTTPS! · A guy with 'Ego' in his name). WebSockets could provide a more robust, full-duplex channel, potentially with standard auth (like API keys or OAuth tokens in the connection URL), but the lack of official documentation signals that the protocol is still maturing its security story. Additionally, because MCP currently uses JSON (a text-based protocol), it inherits all the usual concerns about input validation – developers must be careful to parse and handle JSON-RPC requests properly to avoid injection or crashes in the MCP server.

Supply Chain Risk of Third-Party Tools: Finally, one cannot discuss MCP security without addressing the software supply chain. MCP encourages using third-party tool integrations (often open-source packages or Docker images). Running an MCP server essentially means executing arbitrary code (the code that wraps the tool) on your system, often with significant privileges (e.g., file system or network access if the tool needs it). Thus, installing an unknown MCP server is akin to installing any third-party software with all its attendant risks. The Block security blog points out that *"when users install MCP based extensions they are providing arbitrary code execution privileges to the MCP Server"*, which is a well-understood supply chain problem (Securing the Model Context Protocol | codename goose) (Securing the Model Context Protocol | codename goose). Today, the mitigation relies on classic practices:

- Only use MCP integrations from **trusted sources or vendors** (perhaps those vetted by a community or company).

- Employ code signing and checksums for MCP packages so that consumers can verify they haven't been tampered with (Securing the Model Context Protocol | codename goose).

- In enterprise settings, maintain an **allow-list of approved MCP servers** that employees can use, preventing unvetted code from running in the environment (Securing the Model Context Protocol | codename goose).

However, a more systemic solution might be needed as MCP grows (for example, a central repository of audited MCP integrations or a sandbox mechanism to run untrusted tool code with limited permissions).

Current Status: Summarizing the landscape, MCP's security model as of early 2025 is rudimentary: it assumes a level of trust and pushes responsibility to the transport layer and implementers. There is recognition of these gaps in the community, and initial steps

are being taken to address them (e.g. Cloudflare's integration of OAuth for authorization, discussed below). As one analysis put it, MCP *"isn't quite ready for enterprise prime time"* and *"several pieces of the puzzle need to be solved before [it] becomes a foundational layer of the AI internet"* (MCP: The missing link for agentic AI?). The good news is that those pieces are actively being worked on. The next section explores where MCP is heading and how ongoing efforts are aiming to bolster its security and capabilities.

Future Directions and Ongoing Developments

The rapid adoption of MCP has been accompanied by parallel efforts to **strengthen its security and expand its functionality**. Both the open-source community and companies deploying MCP have proposed enhancements or new standards to address the shortcomings identified above. In this section, we highlight key areas of evolution for MCP: identity and authentication, streaming protocol improvements, OAuth integration, supply chain risk mitigation, and general protocol evolution towards a more mature standard.

Identity and Authentication

One major direction for MCP's future is establishing a robust **identity framework** for agents and users. Traditional web APIs authenticate a user or service via tokens or keys tied to a human identity or a service principal. With agentic AI, there is an additional actor: the agent itself (which might be acting autonomously on a user's behalf). Block's security engineers articulate the need to track a combination of identities: **(1)** the end-user or primary identity (who ultimately authorized this action), **(2)** the agent's identity (which AI or agent instance is making the request), and **(3)** the device or origin of the agent (since an agent could run on a user's device or in cloud) (Securing the Model Context Protocol | codename goose) (Securing the Model Context Protocol | codename goose). Being able to determine this triad – *user + agent + device* – for each action will help companies enforce policies (for example, only a particular trusted agent app is allowed to invoke certain tools on behalf of a user, and only from approved devices or locations) (Securing the Model Context Protocol | codename goose) (Securing the Model Context Protocol | codename goose).

To support this, the MCP specification has been expanding to include standardized **authentication flows**. The MCP spec introduced a draft for OAuth 2.0 integration (as of early 2025) and has since released it in the official spec (Securing the Model Context Protocol | codename goose). The envisioned process is essentially an **OAuth authorization code flow** tailored to MCP:

1. The AI Agent (MCP client) initiates an OAuth login with the MCP server.

2. The MCP server redirects the user to a third-party identity provider (Auth server) for login consent.

3. The user approves and the third-party returns an auth code to the MCP server.

4. The MCP server exchanges the code for an access token from the identity provider.

5. The MCP server then issues its own token or confirmation to the AI agent, allowing it to use the MCP tools under the approved user identity (Securing the Model Context Protocol | codename goose).

This flow is similar to how a user would authorize a third-party app to access their data via APIs, except here the "app" is an AI agent. Cloudflare's managed MCP service has already implemented such an approach: *"People simply sign in and grant permissions to MCP clients using familiar authorization flows,"* explained Rita Kozlov of Cloudflare (MCP: The missing link for agentic AI?). In practice, when an agent wants to use a Cloudflare-hosted MCP server (e.g., one that can access a user's cloud databases), the user will be prompted to log in (perhaps with their OAuth Google/Microsoft account) to grant the agent access. The agent receives a token and includes it in subsequent MCP requests, and the MCP server verifies it before executing any tool action.

Going forward, the **standardization of OAuth** in MCP means agents and servers can converge on common methods of auth, rather than each tool inventing its own. A proposed improvement is to offload more of the OAuth handling to agent platforms themselves (Securing the Model Context Protocol | codename goose). Instead of every MCP server needing to implement a mini web-server for the redirect dance (which is error-prone), agent frameworks could provide a built-in "browser orchestration" to handle user authentication and then pass a token to the MCP server. This would both **streamline integration and reduce security bugs**, as *"having individual MCP implementations implement OAuth themselves is likely to lead to long term security and maintenance issues"* (Securing the Model Context Protocol | codename goose). We can expect MCP libraries and SDKs to increasingly include standard auth modules, and perhaps support for other identity schemes (e.g. API keys, signed JWTs) for non-interactive scenarios.

Streaming Protocol Enhancements

MCP's initial transports (STDIO and HTTP/SSE) have proven adequate for early experimentation, but they are not the pinnacle of modern network protocol design –

especially for the **streaming, bidirectional communication** that agent-tool interactions often require. SSE (Server-Sent Events) allows the server (tool) to stream results to the client (agent), but the communication is still fundamentally **client→server in initiation** (the agent must make an HTTP request and keep it open). In complex multi-step agent workflows, a more flexible channel could be beneficial, for example allowing a tool to notify an agent of an event spontaneously or enabling full **duplex messaging** (both sides sending messages independently). The community has thus been discussing alternatives like **WebSockets or gRPC** for MCP.

WebSockets can provide a persistent, two-way connection over which JSON-RPC messages can flow in both directions. There are indications that some implementations have added WebSocket support (indeed, references to a WebSocket transport exist, although it was not formally documented in the spec as of March 2025 (MCP my HTTPS! · A guy with 'Ego' in his name)). A natural evolution would be for the MCP spec to officially incorporate WebSockets as a transport option, specifying how connection upgrade, authentication, and message framing works in that context.

Beyond WebSockets, some experts suggest leveraging **gRPC** (Google's open-source RPC framework over HTTP/2) or similar modern RPC systems. Sergei Egorov, in his critique "MCP my HTTPS!", argued that much of MCP's functionality could be achieved by simply defining high-level interfaces on top of gRPC, which already supports streaming and has many built-in features (authentication, load balancing, binary encoding, etc.) (MCP my HTTPS! · A guy with 'Ego' in his name). He pointed out that gRPC even has a reflection mechanism that allows clients to discover service methods at runtime (MCP my HTTPS! · A guy with 'Ego' in his name), analogous to how an agent might discover MCP tools. In his view, *"we literally don't need to do anything [new] to allow an LLM to discover defined services, because the wheel was already invented"* in systems like gRPC (MCP my HTTPS! · A guy with 'Ego' in his name). While MCP is not likely to be replaced by gRPC outright (because MCP's value is also in a higher-level convention and growing community support), the **future protocol design could learn from these technologies**. We may see MCP move towards more efficient message encoding (perhaps an optional binary mode instead of JSON) or adopt patterns like **bi-directional streaming** so that agents and tools can maintain a continuous exchange of information rather than the strict call-and-response of JSON-RPC.

In summary, improving MCP's transport layer is on the roadmap. Whether through officially adding WebSocket support, providing a gRPC-based reference implementation, or other means, the goal is to ensure MCP communications are **scalable (low-latency, low-overhead)** and **flexible (supporting real-time, event-driven interactions)**, all without sacrificing security. Any new transport will need thorough review to make sure it includes encryption and authentication hooks from the start, thereby improving on the somewhat barebones security of STDIO/SSE.

Fine-Grained Access Control and Tool Safety

Another future focus is embedding more **policy and safety controls into the MCP standard** itself. As noted, one recent enhancement to the MCP spec is the ability for a tool to declare a **safety classification** for its actions. In the latest release of the protocol, developers can mark a tool or an action as "**readOnly**" or "**destructive**" in its metadata (Securing the Model Context Protocol | codename goose). This metadata is machine-readable by the agent or agent platform. The idea is that an agent can be made to automatically require a secondary confirmation from the user (or some approval mechanism) if a **destructive** action is about to be executed (Securing the Model Context Protocol | codename goose). For example, a tool that can delete data or perform financial transactions might be flagged as destructive, and the agent should then pause and ask "The following action is potentially destructive. Do you want to proceed?" to a human operator or check against a policy rule. This kind of deterministic labeling is a valuable addition on top of purely AI-based safety checks. As the Block team observed, *"having a deterministic aspect to higher risk commands in tandem [with LLM-based checks] ensures good access control"* and provides better protection (Securing the Model Context Protocol | codename goose) (Securing the Model Context Protocol | codename goose). We can expect future versions of MCP to expand such metadata: perhaps categorizing tools by data sensitivity level, required user role, or even adding rate-limiting hints to prevent abuse.

The concept of a **"human in the loop" for dangerous operations** is likely to become a standard practice. MCP could formalize a pattern where certain actions trigger a **notification or approval request** to a human. This aligns with established approaches in DevOps and security (for instance, requiring multi-party approval for deploying to production). One can imagine an enterprise MCP agent that, if instructed to modify access control settings or transfer large funds, automatically generates an approval ticket or message to an administrator for sign-off before proceeding. While not a part of MCP spec today, such workflow integrations will be important for adoption in regulated industries. It is reasonable to foresee extensions or best-practice guides on how to integrate MCP calls with audit logs and approval flows.

OAuth and Authorization Standards

As touched on in the Identity section, OAuth 2.0 integration is on the immediate horizon for MCP. Cloudflare's implementation and the formal spec addition mean that **authorization grants** will be a first-class part of using MCP tools. We will likely see a convention for how MCP servers advertise their required scopes or permissions (similar to how OAuth-based APIs require certain scopes). For example, an MCP server for Google Drive might indicate it needs "Drive: Read/Write" access; when an agent attempts

to connect, the user will be prompted to consent to that scope. The token returned would then be scoped, and the MCP server would only allow corresponding operations.

Another aspect is the **agent identity**: It's one thing to authenticate the human end-user, but ensuring the calling agent is known and trusted is another. We might see mutual authentication schemes where the agent presents its own credential to the MCP server (proving it is a legitimate agent, not malware pretending to be one). This could be achieved with client-side certificates or signed tokens that the agent platform manages. Tying agent identity into the OAuth flow (for instance, including the agent's ID in the OAuth redirect or token claims) could be a way to bind all three identities (user, agent, tool). These are areas of active development and discussion.

Supply Chain Risk Mitigation

In the near future, expect to see more formal solutions for the MCP supply chain issues. Possibilities include:

- **MCP Tool Registries**: A centralized directory of MCP integrations where each entry is vetted or at least community-reviewed. This could be an extension of the official modelcontextprotocol.io site or a third-party service. Having a known-good registry would help users avoid malicious packages.

- **Sandboxing Mechanisms**: Agent runtime environments might start sandboxing MCP servers, especially those fetched from the internet. For example, running each MCP server in a container or with restricted OS permissions (using Linux seccomp, Windows AppContainer, etc.) to limit the blast radius if a tool is compromised. While this is more of an implementation detail than protocol spec, it might be recommended in the MCP documentation for sensitive contexts.

- **Automated Updates and Patches**: As enterprises adopt MCP, they will need ways to keep the integrations up-to-date with security patches. This could lead to standard versioning and update channels for MCP servers, and guidance on how to safely upgrade them without breaking agent workflows.

Block's team recommends basics like only installing trusted MCPs and using integrity checks/signatures (Securing the Model Context Protocol | codename goose), which will likely be supplemented by tooling (for example, an enterprise agent that refuses to load an MCP package unless it's signed by a trusted key). In essence, the same supply chain defenses used for open-source libraries and containers will be applied to MCP, potentially with more urgency given the power an MCP tool might have on a system.

Long-Term Protocol Evolution

Looking further ahead, MCP may evolve from a de-facto community standard into a more formal standard (possibly via a standards body or wide industry coalition) if it continues to gain traction. Its design will iterate as real-world use uncovers edge cases. We've already seen it adapt – e.g., adding OAuth and tool annotations within months of the initial release. Future versions might incorporate:

- **Multi-tenancy support**: allowing one MCP server to serve multiple users securely (Cloudflare's work is a step in this direction (MCP: The missing link for agentic AI?)).

- **Transaction and Payment Safety**: addressing scenarios where agents perform transactions (perhaps integrating with payment APIs with strict verification, to solve the "payment security" problem noted by early users (MCP: The missing link for agentic AI?)).

- **Better Introspection**: tools for an agent to query what capabilities an MCP server provides in a safe way. While basic listing is in the spec, richer descriptions or documentation could be standardized to help agents plan their use of tools.

- **Performance Optimizations**: as usage scales, there may be a push for more efficient encoding than JSON (e.g., binary serialization like MessagePack or ProtoBuf for MCP messages) and batching or multiplexing calls to reduce overhead.

It's also worth noting alternative viewpoints like Egorov's, which challenge whether MCP in its current form should exist at all, or if it should be subsumed by extending existing RPC frameworks (MCP my HTTPS! · A guy with 'Ego' in his name) (MCP my HTTPS! · A guy with 'Ego' in his name). His proposal of a simple extension to gRPC (tagging services as LLM-accessible) is thought-provoking, as it would reuse a decade of prior art (MCP my HTTPS! · A guy with 'Ego' in his name). While MCP's momentum makes it unlikely to be replaced, such critiques will influence MCP's evolution – pushing it to incorporate stronger security and avoid reinventing wheels. Indeed, many of the improvements discussed (authentication, streaming, introspection) are about bringing MCP to feature-parity with traditional API technologies, while maintaining its focus on AI agent use-cases.

As Yoko Li wrote, *"if done right, MCP could become the default interface for AI-to-tool interactions and unlock a new generation of autonomous, multi-modal, and deeply integrated AI experiences."* (MCP: The missing link for agentic AI?) That encapsulates the

long-term vision: MCP or something like it becomes a foundational layer of the AI ecosystem, much as HTTP/REST did for web services. To reach that point, the ongoing efforts in identity, security, and robustness are not just ancillary – **they are essential**. Early adopters like Cloudflare and Microsoft contributing back solutions (OAuth flows, hosting infrastructure) and security researchers sharing learnings (like Block's team, or community critiques) indicate a healthy trajectory of improvement.

Conclusion

Model Context Protocol stands out as a promising solution to a pressing need in AI: enabling **secure, scalable, and standardized interactions between autonomous AI agents and the vast world of software tools and data sources**. In this white paper, we examined how MCP works and the advantages it offers – from making tool use deterministic and reliable, to fostering an ecosystem of easily pluggable AI integrations. We also took a hard look at MCP's current security model, finding that while the foundations are in place for basic use, significant gaps remain in authentication, authorization, and safe deployment practices. These gaps are not being ignored: the MCP community and industry supporters are actively working on enhancements such as OAuth-based authentication flows, richer permission models, safer transport protocols, and supply chain protections.

MCP's journey is reminiscent of early web API standards – starting simple to achieve adoption, then layering on security and sophistication as usage matures. It has already begun to incorporate feedback (e.g., adding identity features and tool safety annotations) and will need to continue evolving. For engineers and security professionals, the key is to engage with MCP's development now: contribute to its open specifications, build secure implementations, and apply sound security engineering around it (from code auditing new MCP servers to sandboxing and monitoring agent behaviors). By doing so, we can ensure that the powerful capabilities unlocked by agentic AI **do not come at the expense of security or control**.

In conclusion, MCP is both a technical and a socio-technical artifact. Technically, it bridges AI and software in a standard way; socio-technically, it represents a collaborative movement (across companies and open-source communities) to shape how future AI systems will safely extend their reach. The coming year or two will be critical as MCP transitions from an experimental "cool possibility" to a hardened infrastructure component. If successful, we may soon take for granted that any AI agent, regardless of who built it, can interface with any tool under consistent security guardrails – a true "USB-C for AI" realized (MCP my HTTPS! · A guy with 'Ego' in his name). Achieving that will require continued diligence in protocol security, thoughtful extension of its features, and perhaps a bit of wisdom from past protocols to avoid repeating old mistakes. The

work is ongoing, but the direction is clear: **MCP is evolving to meet its grand promise of enabling the next generation of integrated, autonomous AI – safely and at scale.**

Sources

- Alex Rosenzweig *et al.*, "Securing the Model Context Protocol," *Block Engineering Blog*, March 31, 2025 (Securing the Model Context Protocol | codename goose) (Securing the Model Context Protocol | codename goose).

- Tom Krazit, "MCP: The Missing Link for Agentic AI?" *Runtime Newsletter*, March 25, 2025 (MCP: The missing link for agentic AI?) (MCP: The missing link for agentic AI?).

- Sergei Egorov, "MCP My HTTPS!" personal blog, March 12, 2025 (MCP my HTTPS! · A guy with 'Ego' in his name) (MCP my HTTPS! · A guy with 'Ego' in his name).